JICSCI803
Algorithms and Data Structures
March to June 2019

Highlights of Lecture 12

Parallel Algorithm
More Graph Algorithm

Parallel Algorithms

– Everything we have looked at so far has been based on one key assumption:

• Each instruction in an algorithm is performed one-at-a-time.

• In other words, all of the algorithms we have looked at are sequential.

Types of Parallelism

– We can classify parallel architectures by looking at the number of different instruction and data streams available in the architecture.

- SIMD (Single instruction, Multiple data) –Array/Vector processors.
  - Each processor performs the same instruction on a different data element.
- MISD (Multiple instruction, Single data) –Pipeline processors.
  - Each processor performs a different instruction on a single data stream.
- MIMD (Multiple instruction, Multiple data) –high-level parallel processors.
  - Each processor operates fully independently of each of the others.

Types of Parallelism

– Consider the following code:

```
for i = 1, to 10 do
    if i is odd then
        x[i] = i
    else
        x[i] = 2 * I
```

–How would we code this for parallel execution?

MIMD

```
for i = 1, to 10 do in parallel
   if i is odd then
      x[i] = i
   else
      x[i] = 2 * I
```

– In this case each processing element performs one of the two possible calculations x[i] = i or x[i] = 2 * i.

– This is possible because the MIMD architecture allows processors to execute different instructions at the same time.

SIMD

```
for i = 1, to 10 step 2 do in parallel
        x[i] = i
    for i = 2, to 10 step 2 do in parallel
        x[i] = 2 * i
```

– In this case each processing element performs the same calculation at each step first x[i] = i and then x[i] = 2 * i.

– This is necessary because the SIMD architecture requires processors to execute the same instructions at the same time.

Parallel Sorting

– Consider the following procedure:

```
program MISDsort
  get max
  repeat
    get n
    if n < 0 then
       put max
    elseif n > max then
       put max
       max = n
    else
       put n
 until n < 0
 put –1
```

Parallel Sorting

– Let us feed it the sequence 5 7 9 2 3 4 1 6 –1
– What is the output?

Parallel Sorting

– 5 7 9 2 3 4 1 6 –1

input	max	output
5	5	

Parallel Sorting

– 5 7 9 2 3 4 1 6 –1

input	max	output
5	5	
7	7	5

Parallel Sorting

– 5 7 9 2 3 4 1 6 –1

input	max	output
5	5	
7	7	5
9	9	7

Parallel Sorting

– 5 7 9 2 3 4 1 6 –1

input	max	output
5	5	
7	7	5
9	9	7
2	9	2

Parallel Sorting

– 5 7 9 2 3 4 1 6 –1

input	max	output
5	5	
7	7	5
9	9	7
2	9	2
3	9	3

Parallel Sorting

– 5 7 9 2 3 4 1 6 –1

input	max	output
5	5	
7	7	5
9	9	7
2	9	2
3	9	3
4	9	4

Parallel Sorting

– 5 7 9 2 3 4 1 6 –1

input	max	output
5	5	
7	7	5
9	9	7
2	9	2
3	9	3
4	9	4
1	9	1

Parallel Sorting

– 5 7 9 2 3 4 1 6 –1

input	max	output
5	5	
7	7	5
9	9	7
2	9	2
3	9	3
4	9	4
1	9	1
6	9	6

Parallel Sorting

– 5 7 9 2 3 4 1 6 –1

input	max	output
5	5	
7	7	5
9	9	7
2	9	2
3	9	3
4	9	4
1	9	1
6	9	6
– 1	9	9

Parallel Sorting

– 5 7 9 2 3 4 1 6 –1

input	max	output
5	5	
7	7	5
9	9	7
2	9	2
3	9	3
4	9	4
1	9	1
6	9	6
– 1	9	9
		– 1

Parallel Sorting

Input 5 7 **9** 2 3 4 1 6 -1

Output 5 7 2 3 4 1 6 9-1

- What has this process done?
- It has delayed the output of the largest element until last.
- Now consider what happens if we pipe the output of such a program into another copy of the same program.

Parallel Sorting

–Input 5 7 9 2 3 4 1 6 –1

in	max	out		in	max	out
5	5					

Parallel Sorting

–Input 5 7 9 2 3 4 1 6 –1

in	max	out		in	max	out
5	5					
7	7	5		5	5	

Parallel Sorting

–Input 5 7 9 2 3 4 1 6 –1

in	max	out		in	max	out
5	5					
7	7	5		5	5	
9	9	7		7	7	5

Parallel Sorting

–Input 5 7 9 2 3 4 1 6 –1

in	max	out		in	max	out
5	5					
7	7	5		5	5	
9	9	7		7	7	5
2	9	2		2	7	2

Parallel Sorting

–Input 5 7 9 2 3 4 1 6 –1

in	max	out		in	max	out
5	5					
7	7	5		5	5	
9	9	7		7	7	5
2	9	2		2	7	2
3	9	3		3	7	3

Parallel Sorting

–Input 5 7 9 2 3 4 1 6 –1

in	max	out		in	max	out
5	5					
7	7	5		5	5	
9	9	7		7	7	5
2	9	2		2	7	2
3	9	3		3	7	3
4	9	4		4	7	4

Parallel Sorting

–Input 5 7 9 2 3 4 1 6 –1

in	max	out		in	max	out
5	5					
7	7	5		5	5	
9	9	7		7	7	5
2	9	2		2	7	2
3	9	3		3	7	3
4	9	4		4	7	4
1	9	1		1	7	1

Parallel Sorting

–Input 5 7 9 2 3 4 1 6 –1

in	max	out		in	max	out
5	5					
7	7	5		5	5	
9	9	7		7	7	5
2	9	2		2	7	2
3	9	3		3	7	3
4	9	4		4	7	4
1	9	1		1	7	1
6	9	6		6	7	6

Parallel Sorting

–Input 5 7 9 2 3 4 1 6 –1

in	max	out		in	max	out
5	5					
7	7	5		5	5	
9	9	7		7	7	5
2	9	2		2	7	2
3	9	3		3	7	3
4	9	4		4	7	4
1	9	1		1	7	1
6	9	6		6	7	6
– 1	9	9		9	9	7

Parallel Sorting

–Input 5 7 9 2 3 4 1 6 –1

in	max	out		in	max	out
5	5					
7	7	5		5	5	
9	9	7		7	7	5
2	9	2		2	7	2
3	9	3		3	7	3
4	9	4		4	7	4
1	9	1		1	7	1
6	9	6		6	7	6
– 1	9	9		9	9	7
		– 1		– 1		9

Parallel Sorting

–Input 5 7 9 2 3 4 1 6 –1

in	max	out		in	max	out
5	5					
7	7	5		5	5	
9	9	7		7	7	5
2	9	2		2	7	2
3	9	3		3	7	3
4	9	4		4	7	4
1	9	1		1	7	1
6	9	6		6	7	6
– 1	9	9		9	9	7
		– 1		– 1		9
						– 1

Parallel Sorting

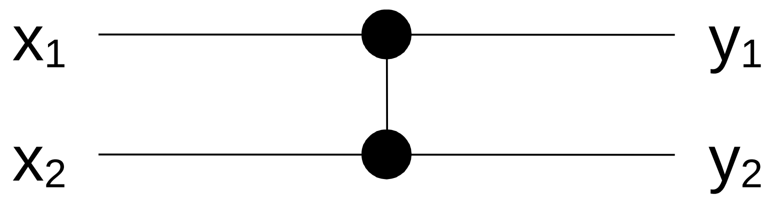
Input 5 7 9 2 3 4 1 6 -1

Output 5 2 3 4 1 6 7 9 -1

- Now the last two elements are in order.
- If we combine n - 1 copies of the program in sequence we have a system to sort an n-element array.
- If we run the copies in parallel we have a sort algorithm which O(n).
- This mechanism of piping data through a series of parallel processes in sequence is the basis of MISD parallelism.

Parallel Sorting

– Consider the following processing element:

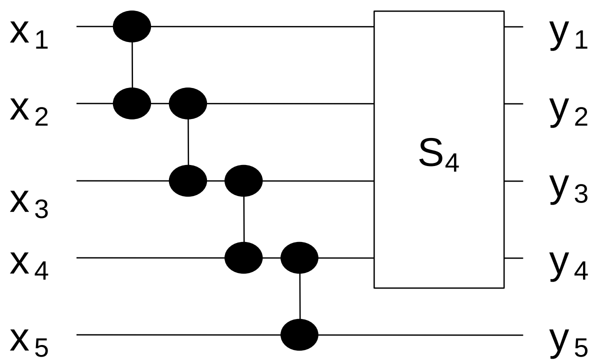


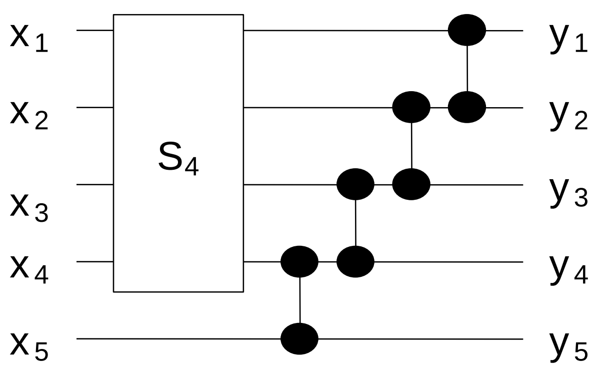
– $y_1 = \min(x_1, x_2)$
– $y_2 = \max(x_1, x_2)$
– This is a comparator.
– Can we combine comparators in parallel to sort a set of numbers?

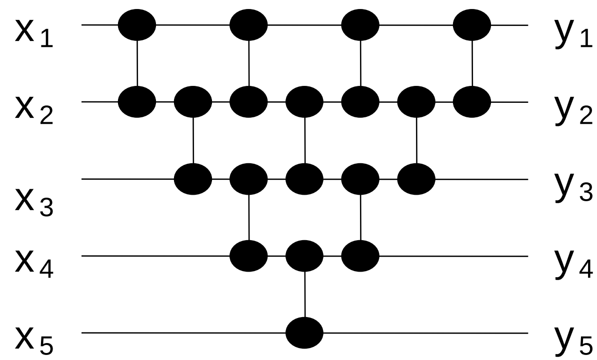
Parallel Sorting

– If we have a circuit S_n *which sorts an n-element set of numbers can we combine it with some additional comparators to produce* S_{n+1}*?*

Parallel Sorting



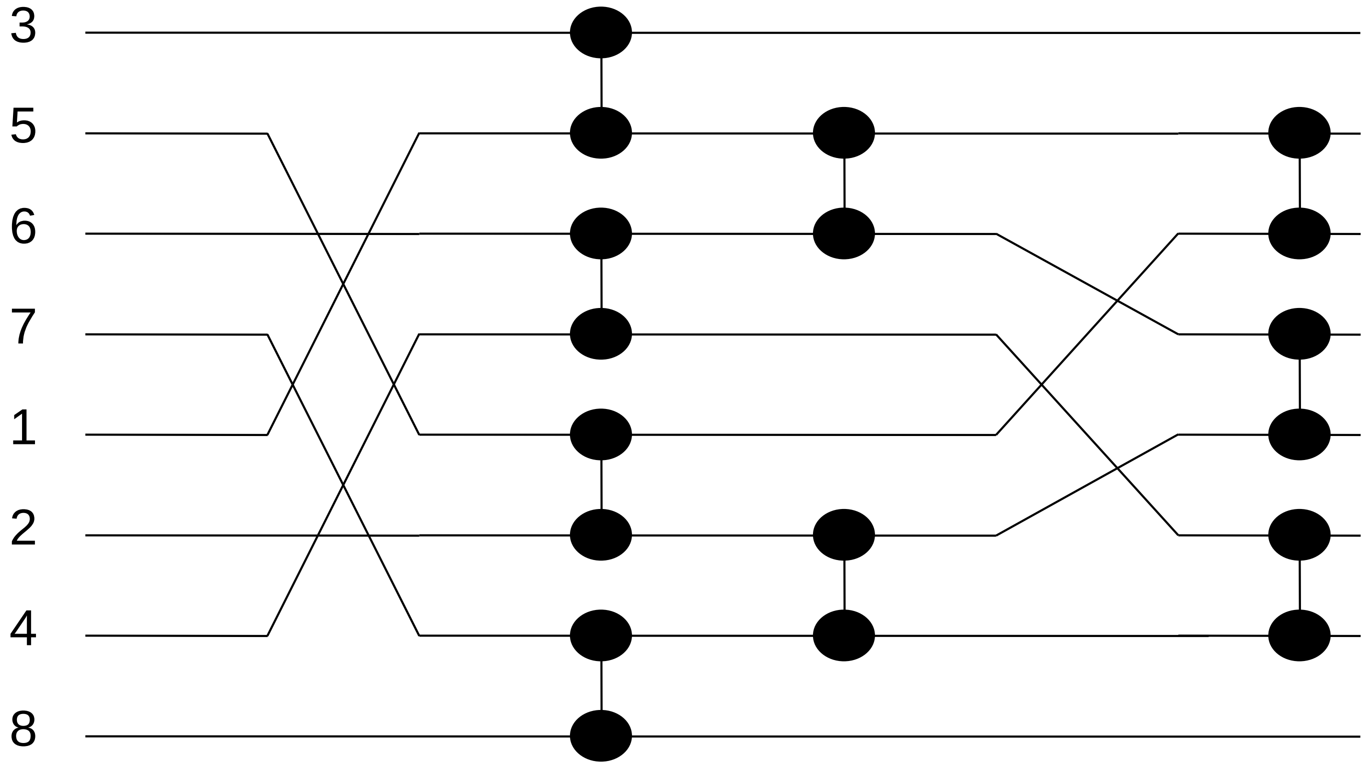




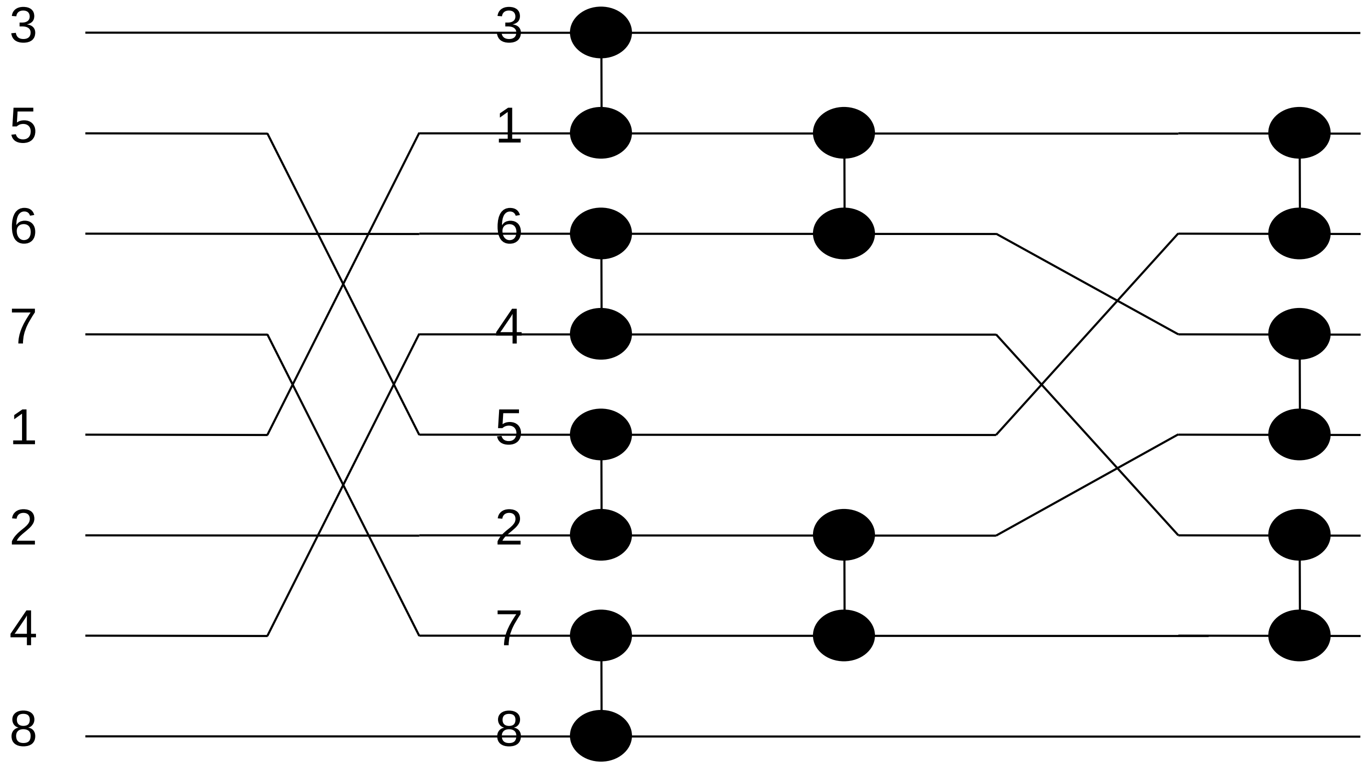
Parallel Sorting by Merging

- Suppose we have two sorted streams, each of n elements.
- Can we use a method analogous to mergesort in parallel?

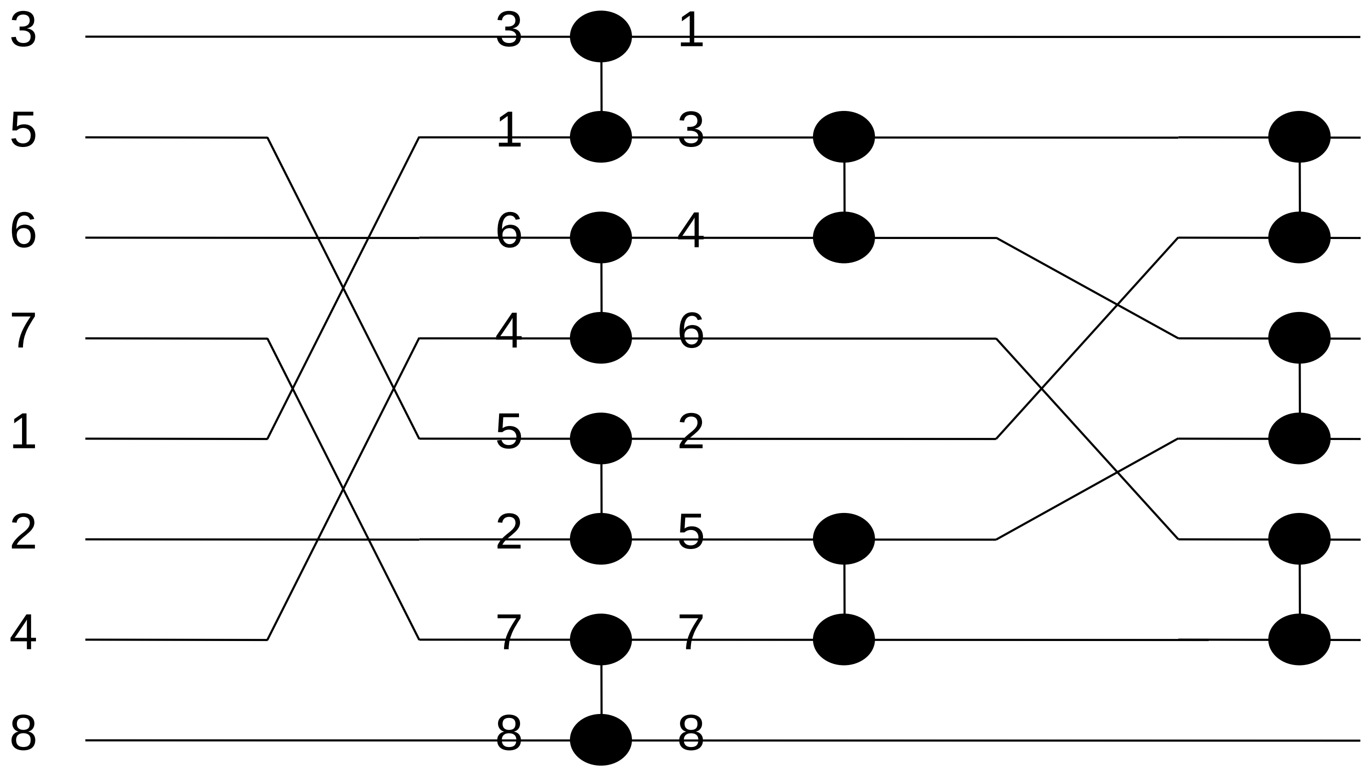
Parallel Sorting by Merging



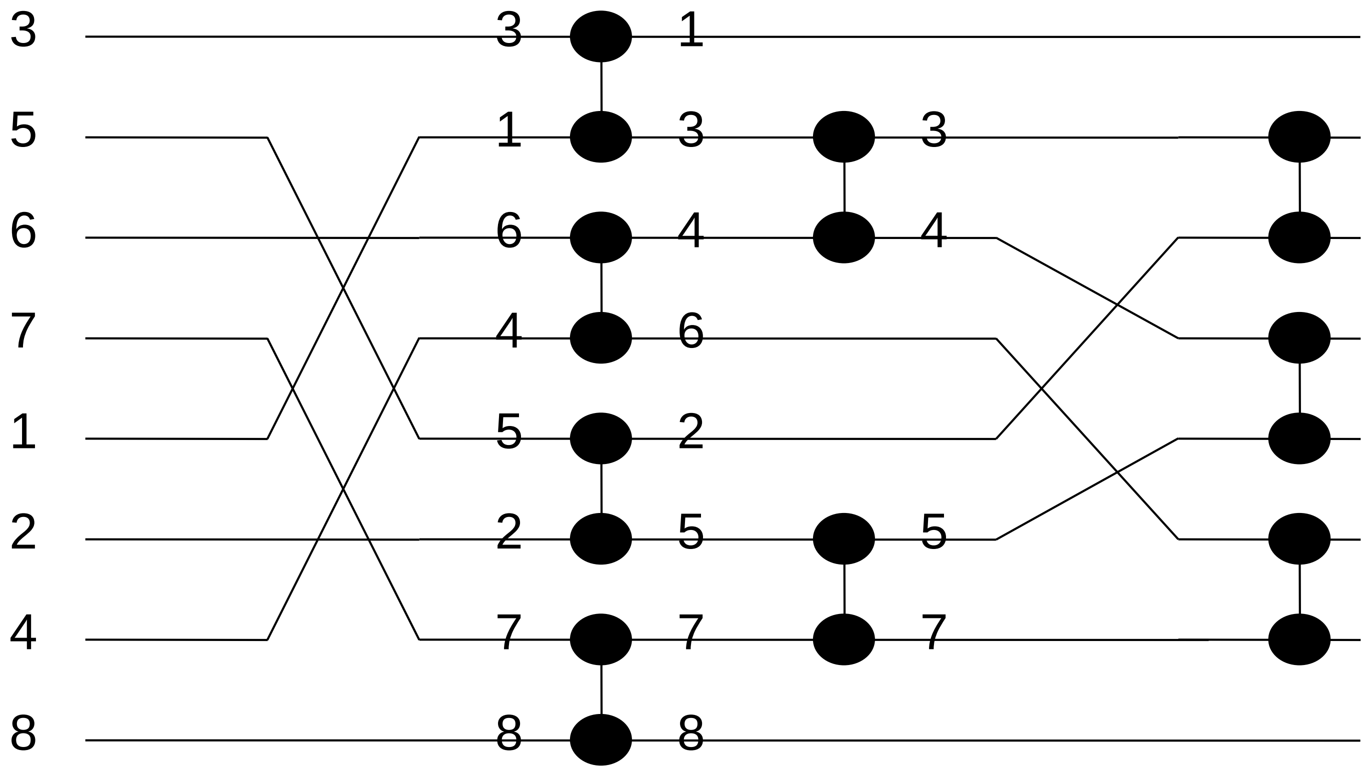
Parallel Sorting by Merging



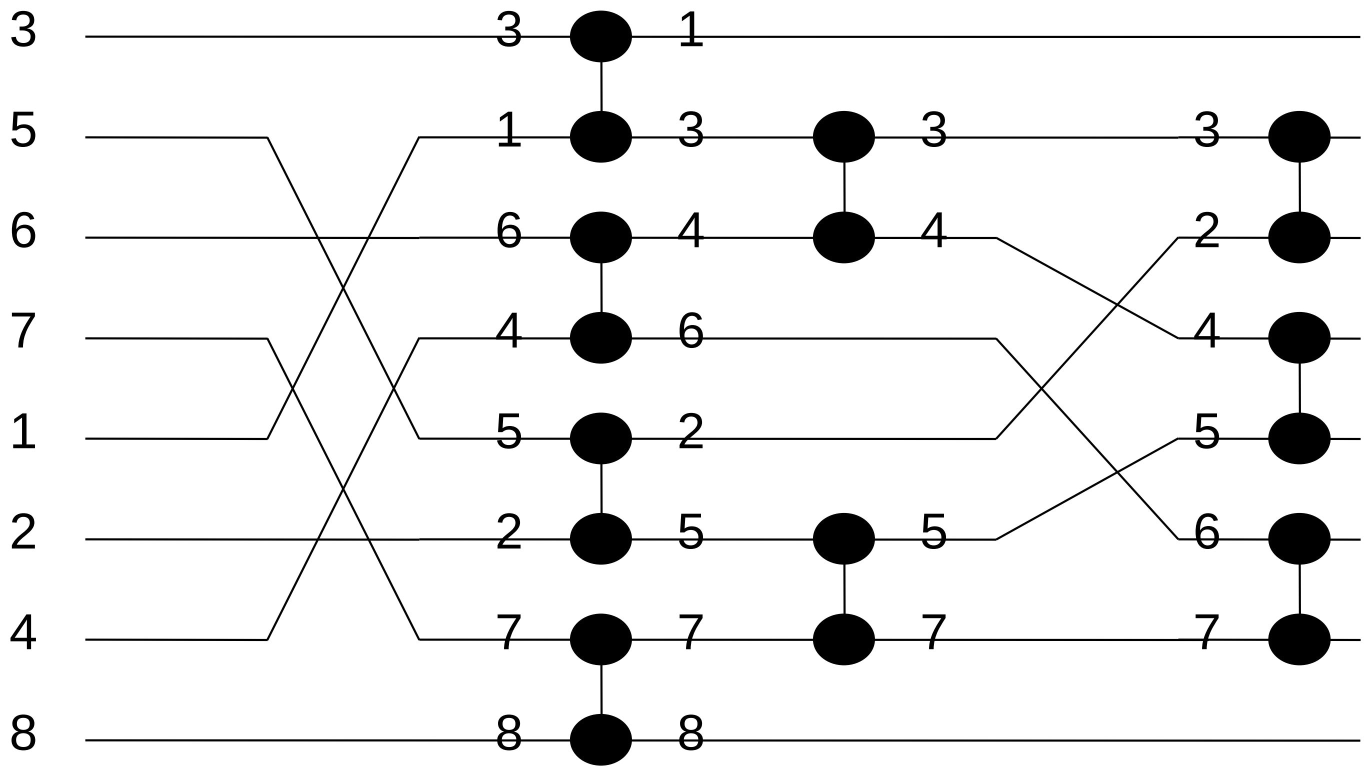
Parallel Sorting by Merging



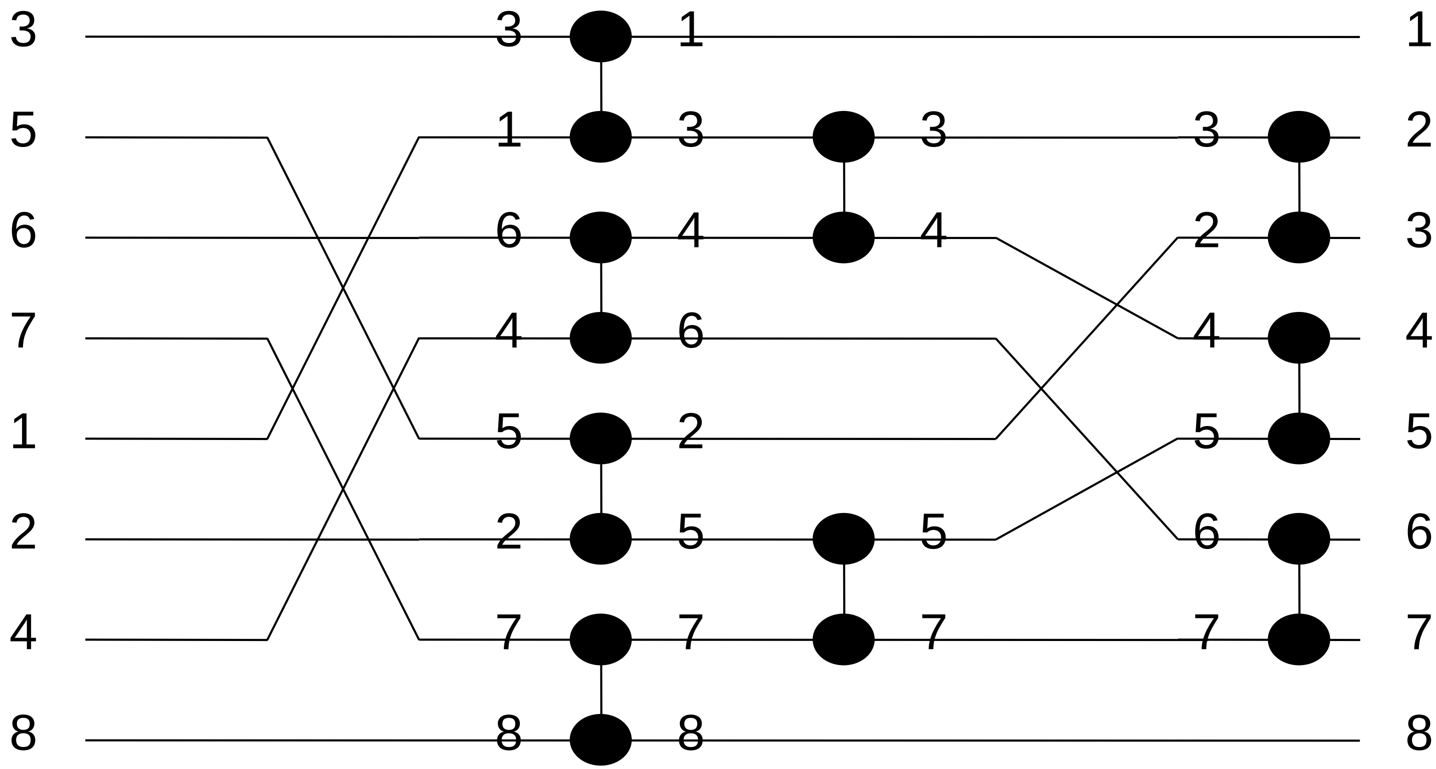
Parallel Sorting by Merging



Parallel Sorting by Merging



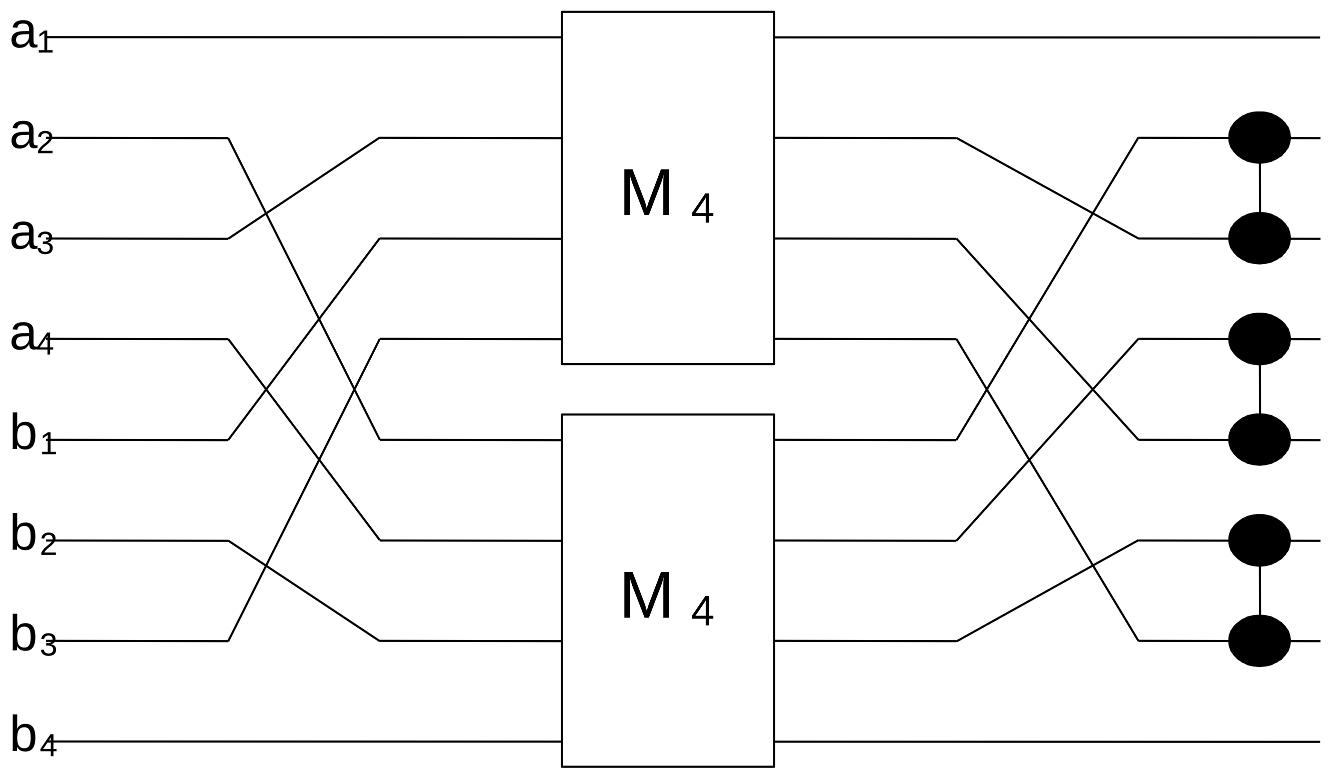
Parallel Sorting by Merging



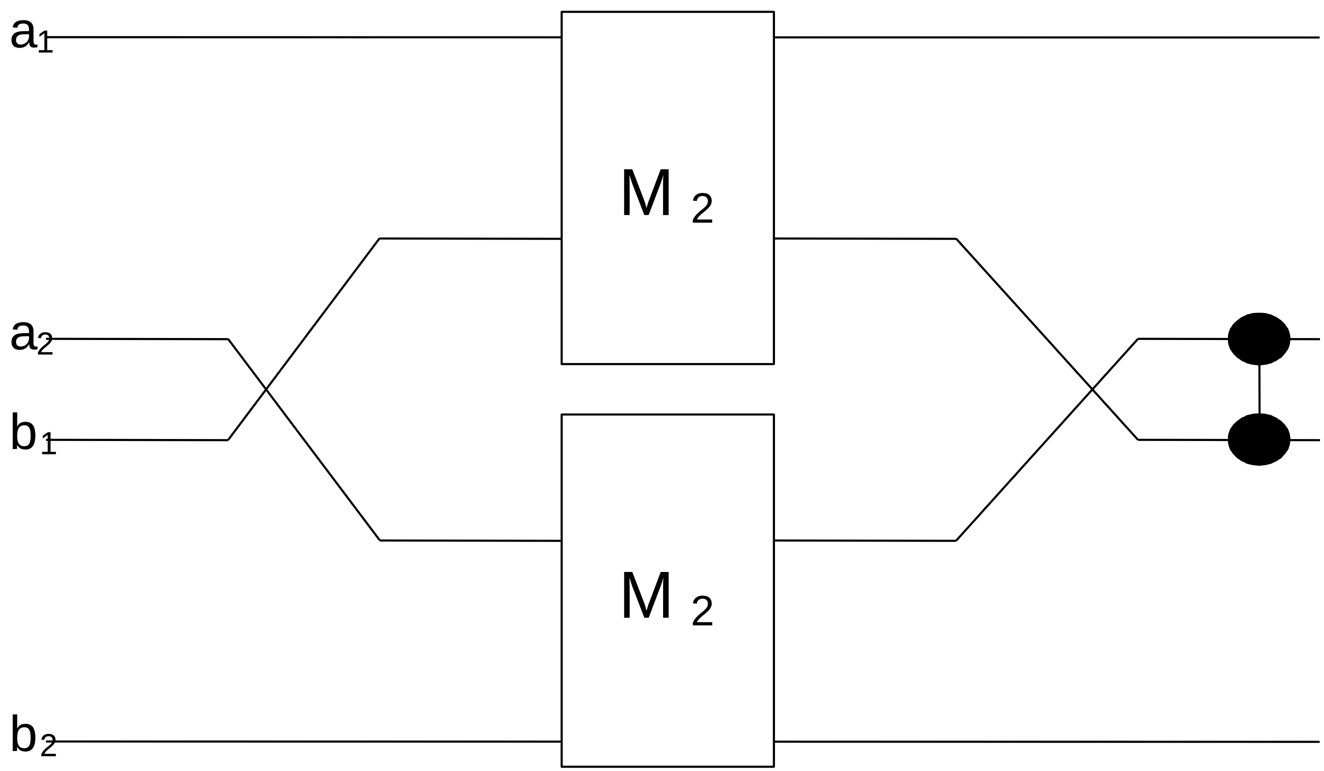
Parallel Sorting by Merging

- Suppose we have two sorted streams, each of n elements.
- Can we use a method analogous to mergesort in parallel?
- Let us assume that we have a circuit that merges two n/2 element streams.

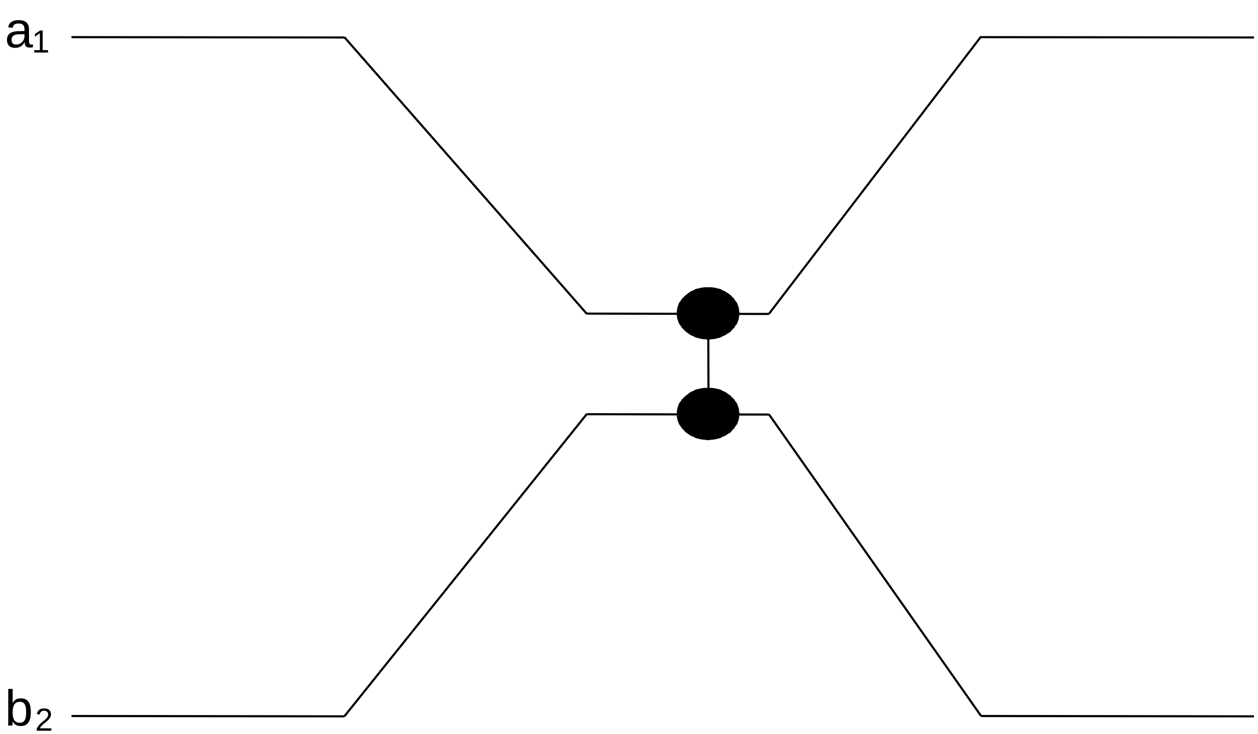
Parallel Sorting by Merging –M_8



Parallel Sorting by Merging –M_4

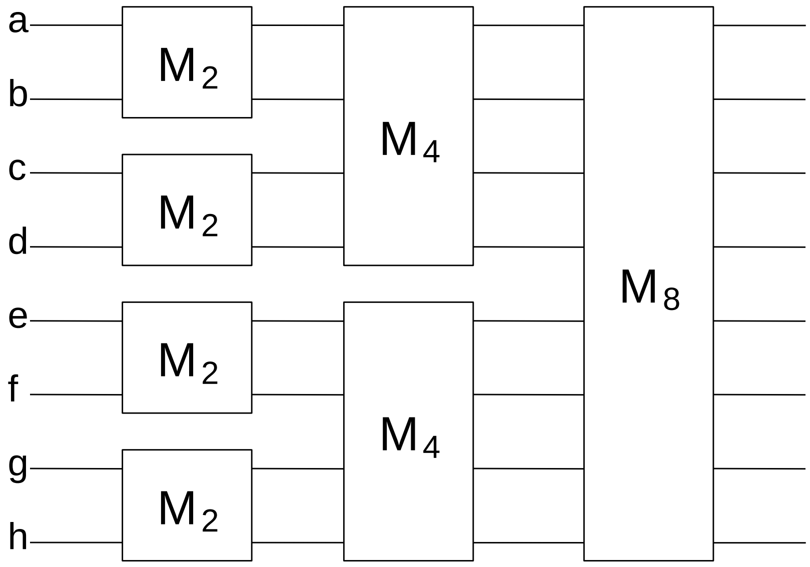


Parallel Sorting by Merging –M_2



Parallel Sorting by Merging

– We can combine these circuits to sort a parallel stream of numbers



More About Graphs

More About Graphs

- A great range of problems can be expressed in terms of graphs.
- We have already seen some of these:
  - Minimal spanning tree
  - Shortest path
- So far, all the algorithms we have examined have imposed an order on the nodes or the edges.
- This is not always necessary.

Graphs and Games

– Consider the following game:

- Initially there is a heap of n matches between two players.
- The first player may remove as many matches as she likes between 1 and n–1.
- Thereafter each player can remove between 1 and twice the number their opponent just took.
- The winner is the person who removes the last match.

The Match Game

– Consider the following situation:
  - There is a pile of five matches in front of you and your opponent has just taken two matches. (the original pile has seven matches)
    - Therefore, you can take 1, 2, 3 or 4 matches but not 5.
  - What should your next move be?
    - If you take 2, 3, or 4 matches, your opponent can win next turn.
    - So you should take 1 match.
  - But what about more complex positions?

The Match Game as a graph

- We can represent positions in the match game as nodes on a graph.
- We can connect nodes to show sequences of possible moves.
- We label each node with two numbers:
  - The number of matches left at this stage of the game,
  - The number of matches which can be removed.
- Clearly node 0, 0 represents the end of the game.

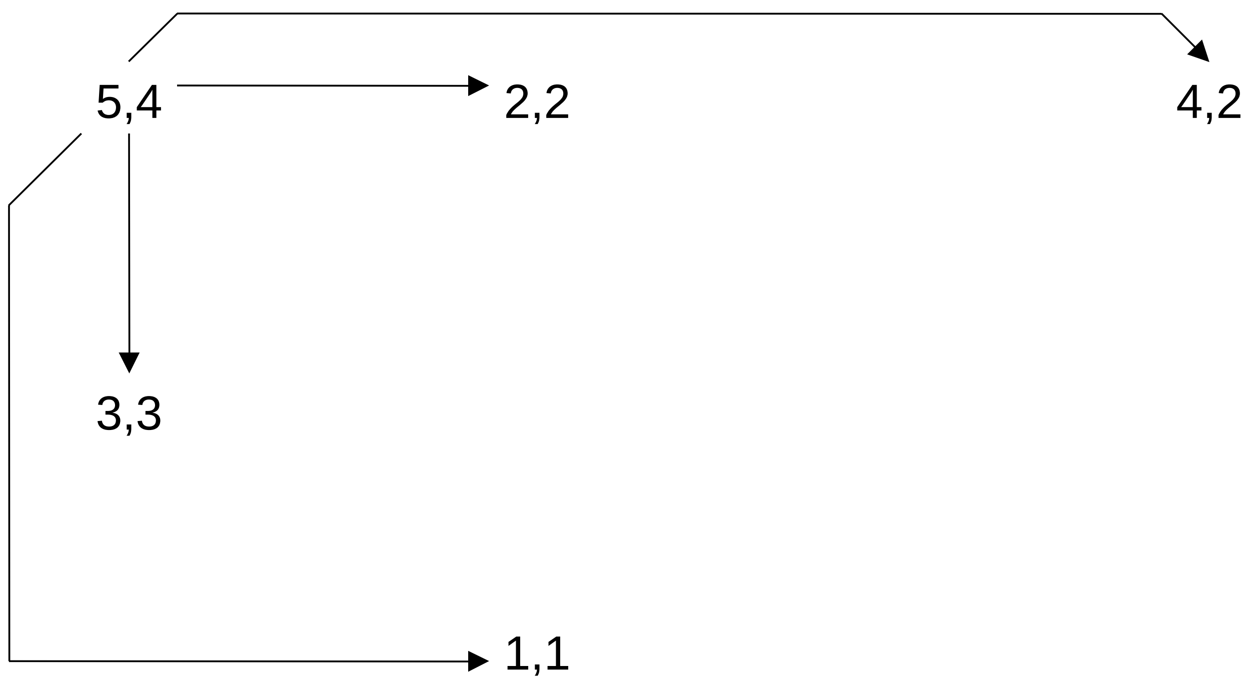
The Match Game as a graph

- Let losing nodes be squares and winning nodes be circles.
- Clearly 0, 0 is a losing node.
- 0, 0 can be reached from any node n, n.
- Any path leading to a winning node is a bad move.
- Any path leading to a losing node is a good move.
- Let us colour good moves in green.
- Let us colour bad moves in red.

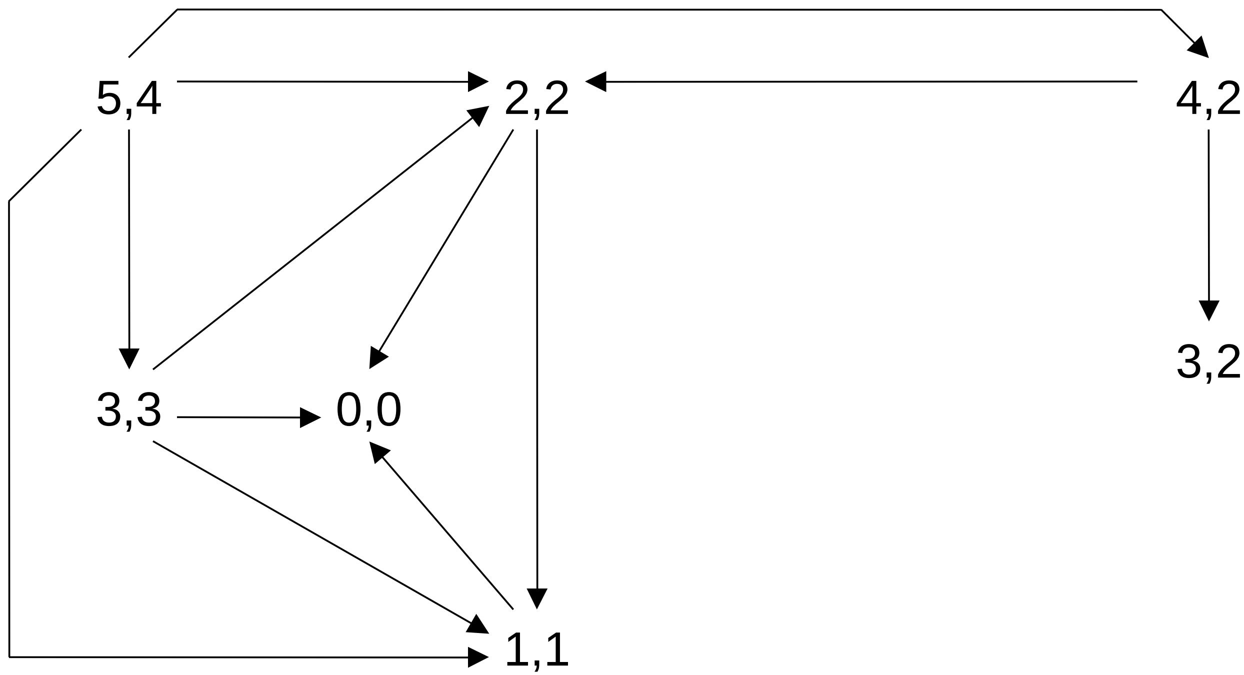
Building the Match Game graph for 5, 4

5,4

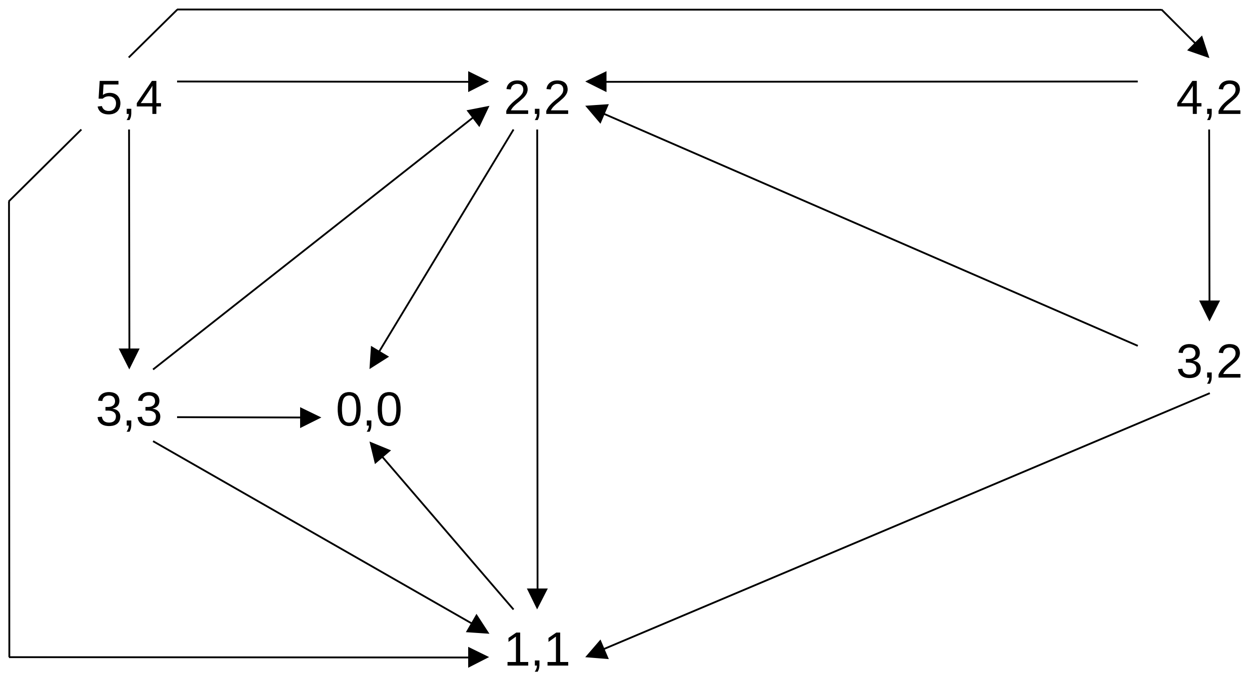
Building the Match Game graph for 5, 4



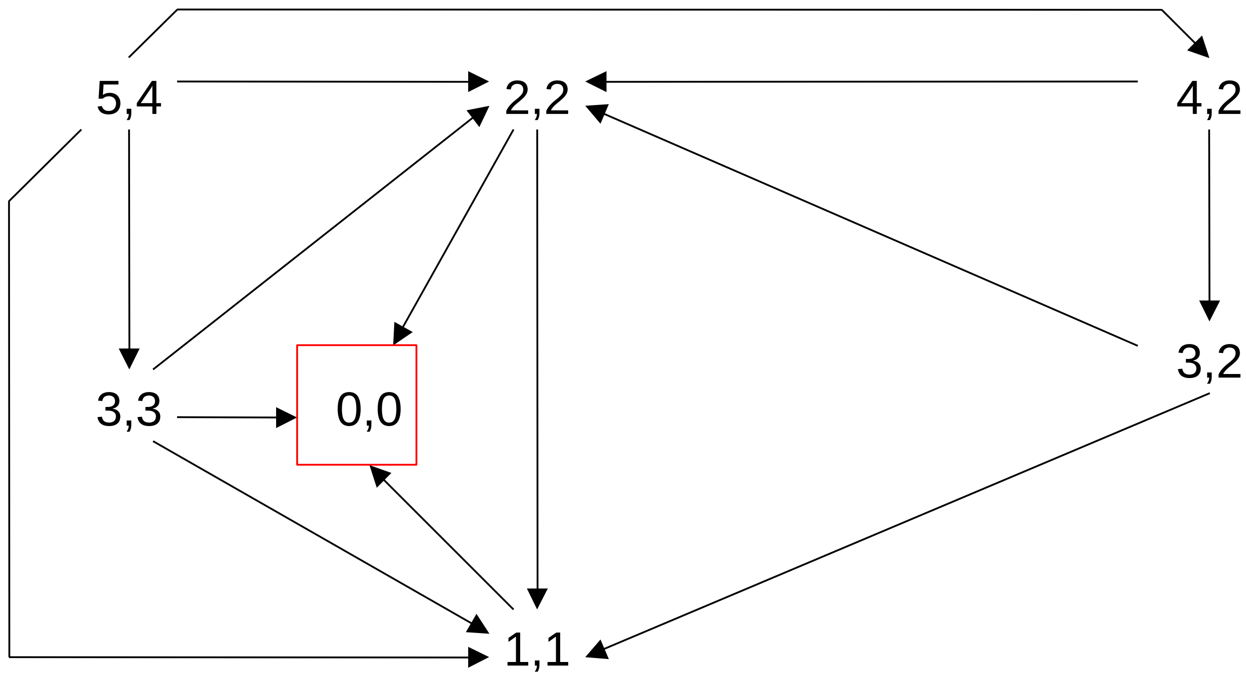
Building the Match Game graph for 5, 4



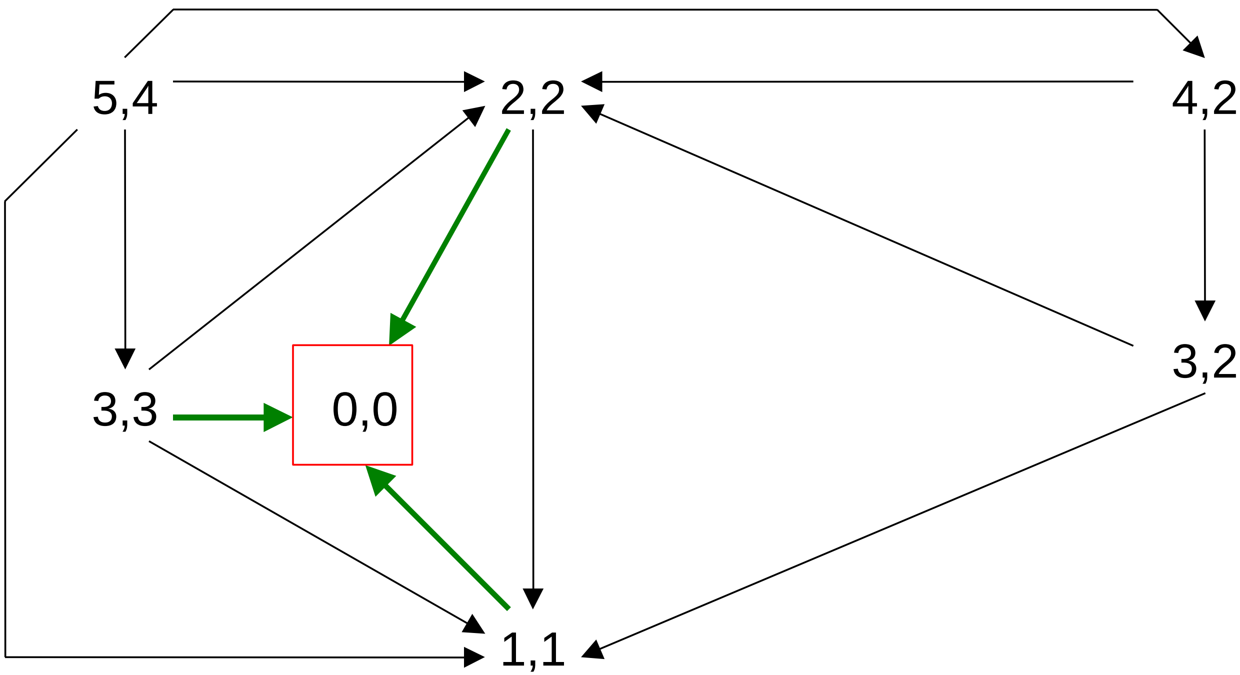
Building the Match Game graph for 5, 4



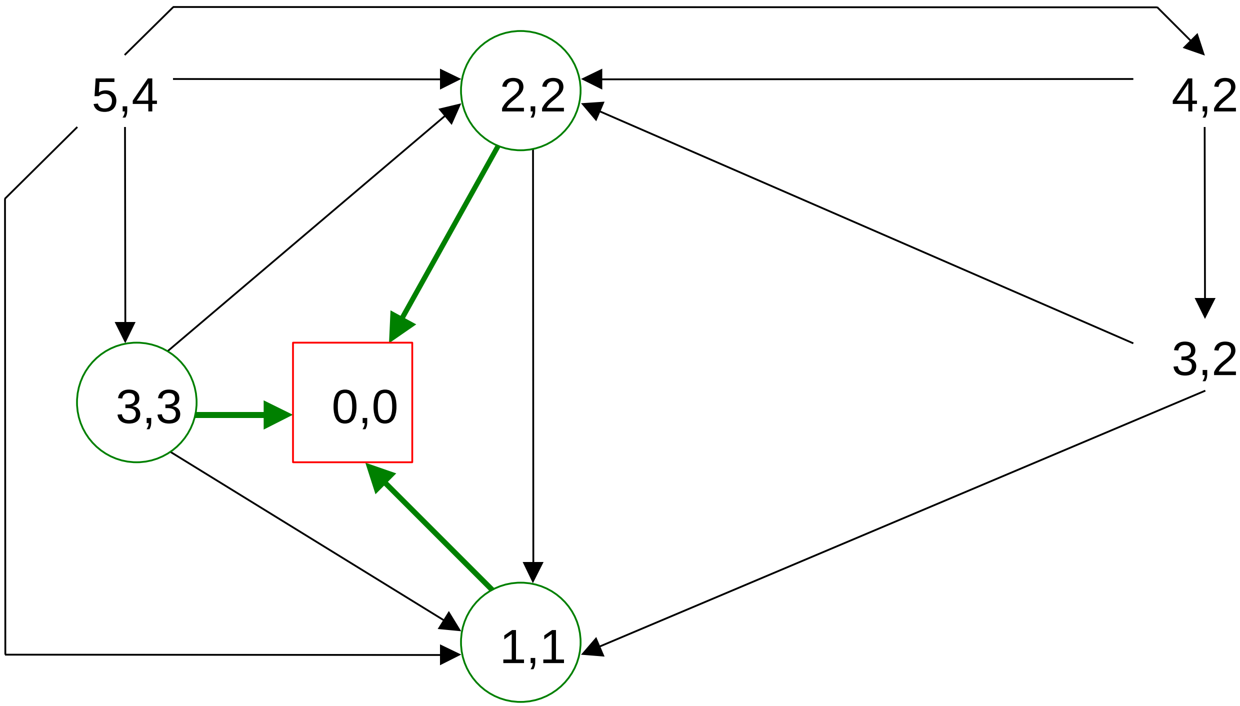
Analysing the Match Game graph for 5, 4



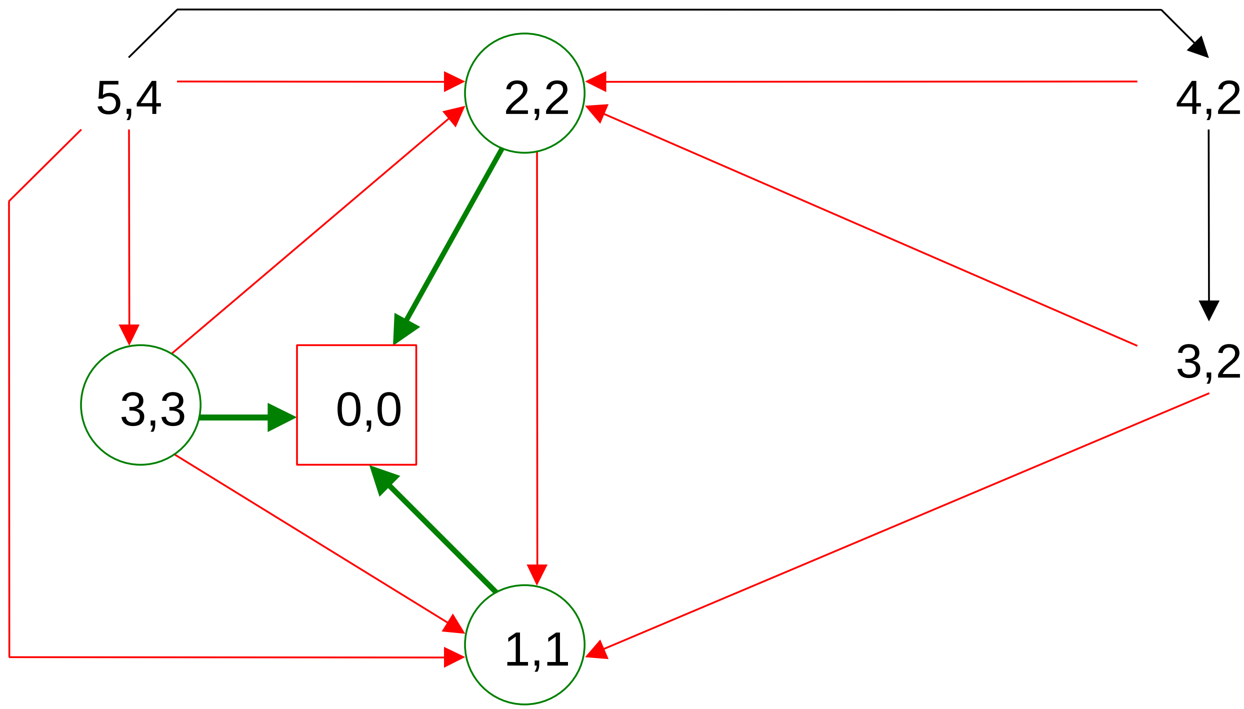
Analysing the Match Game graph for 5, 4



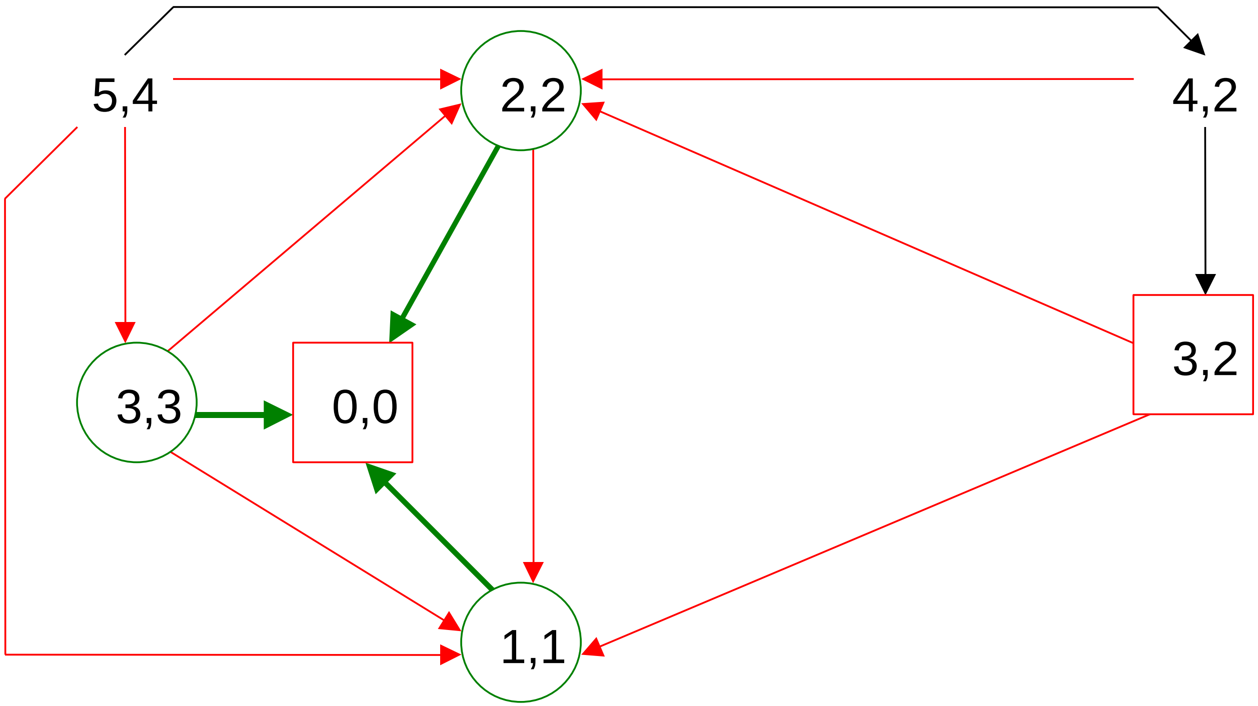
Analysing the Match Game graph for 5, 4



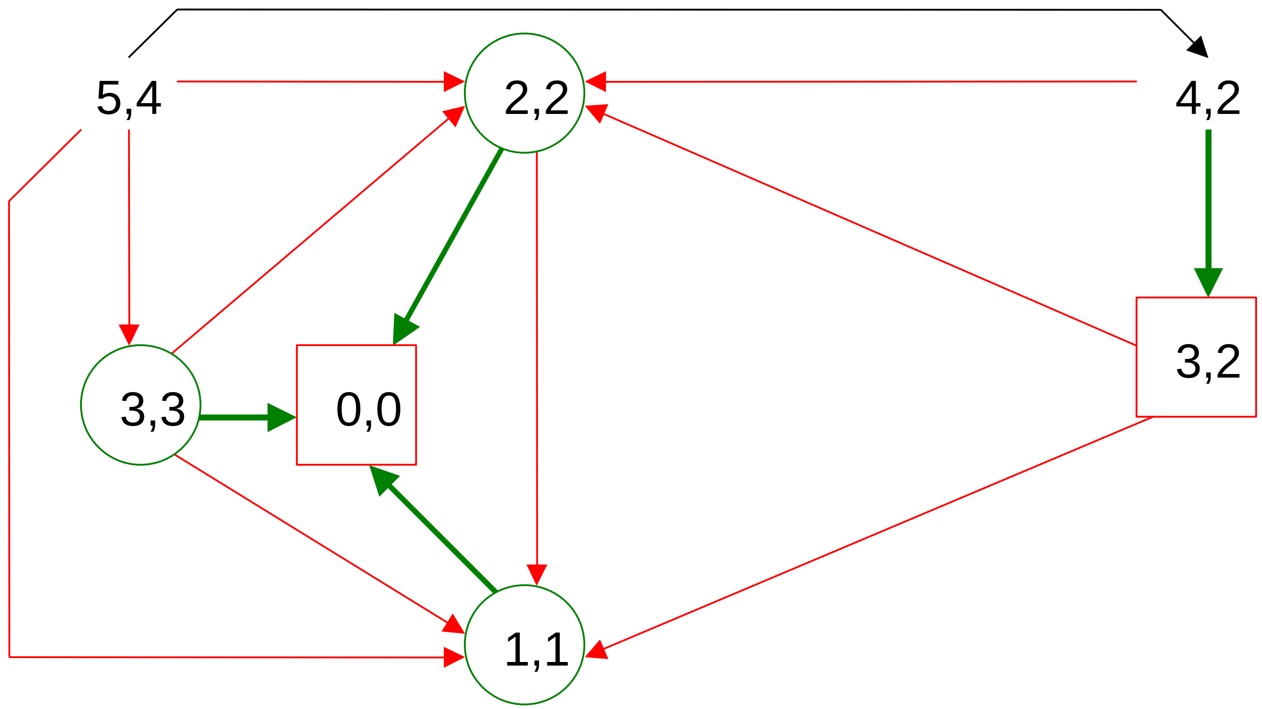
Analysing the Match Game graph for 5, 4



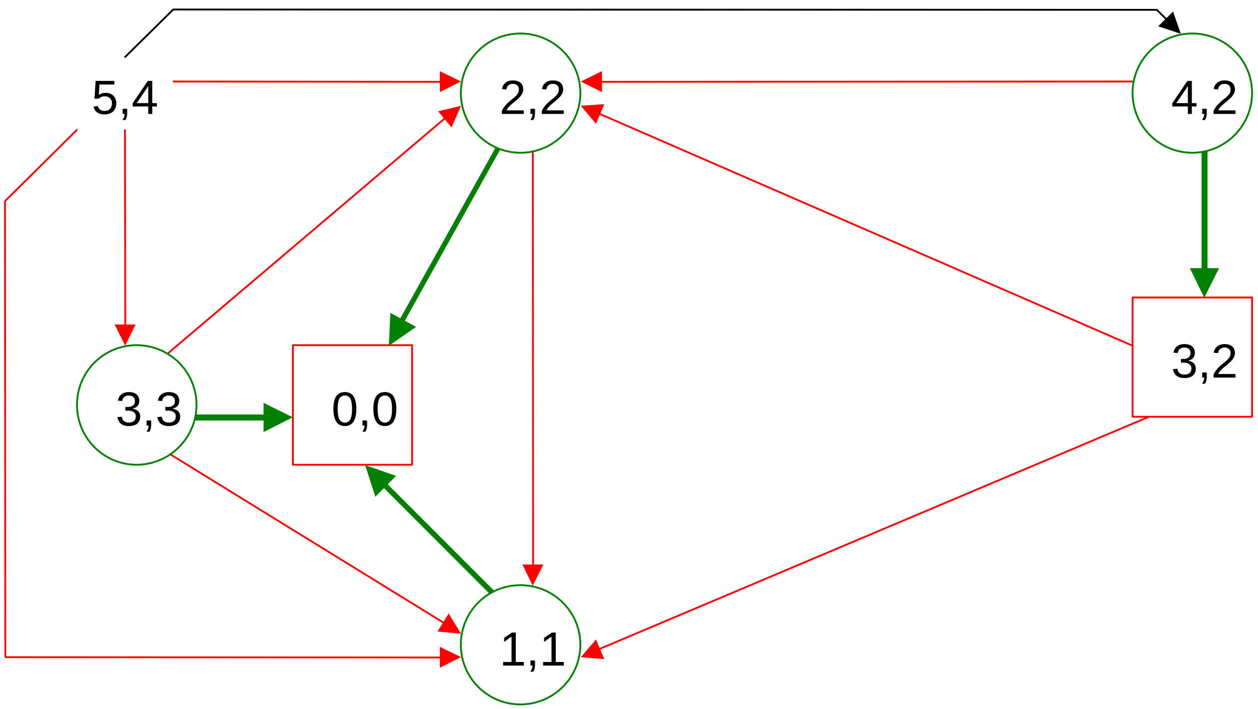
Analysing the Match Game graph for 5, 4



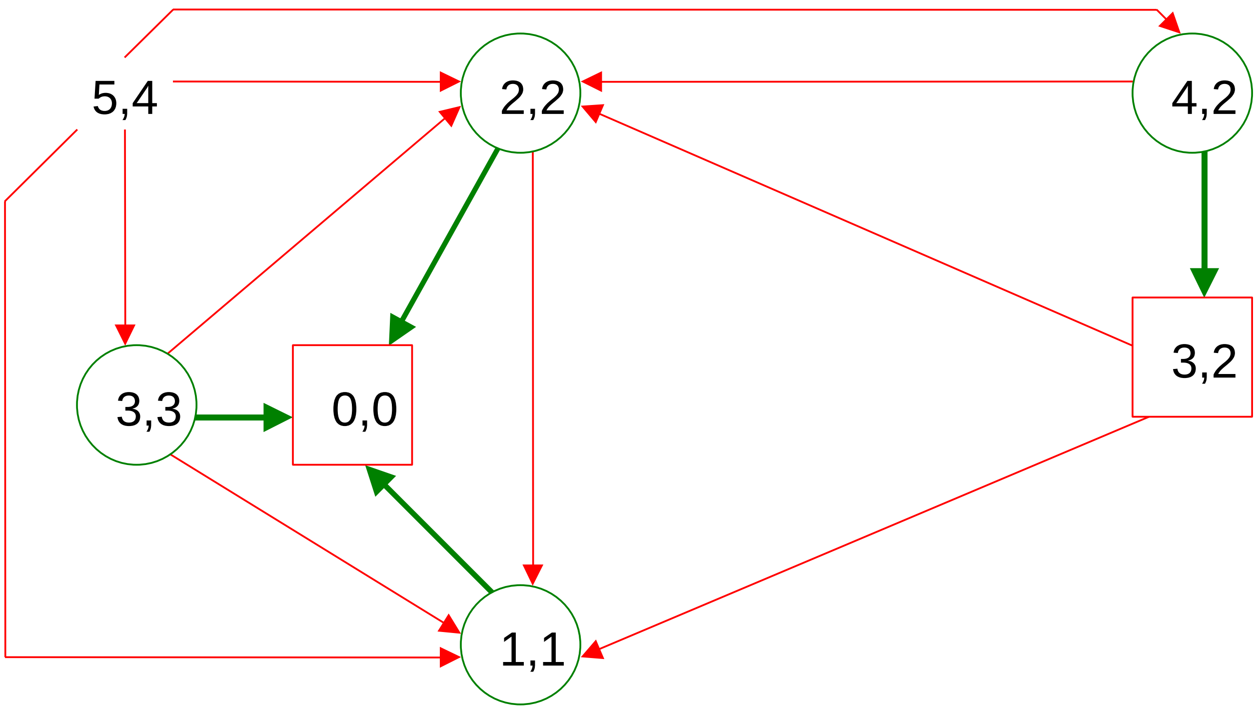
Analysing the Match Game graph for 5, 4



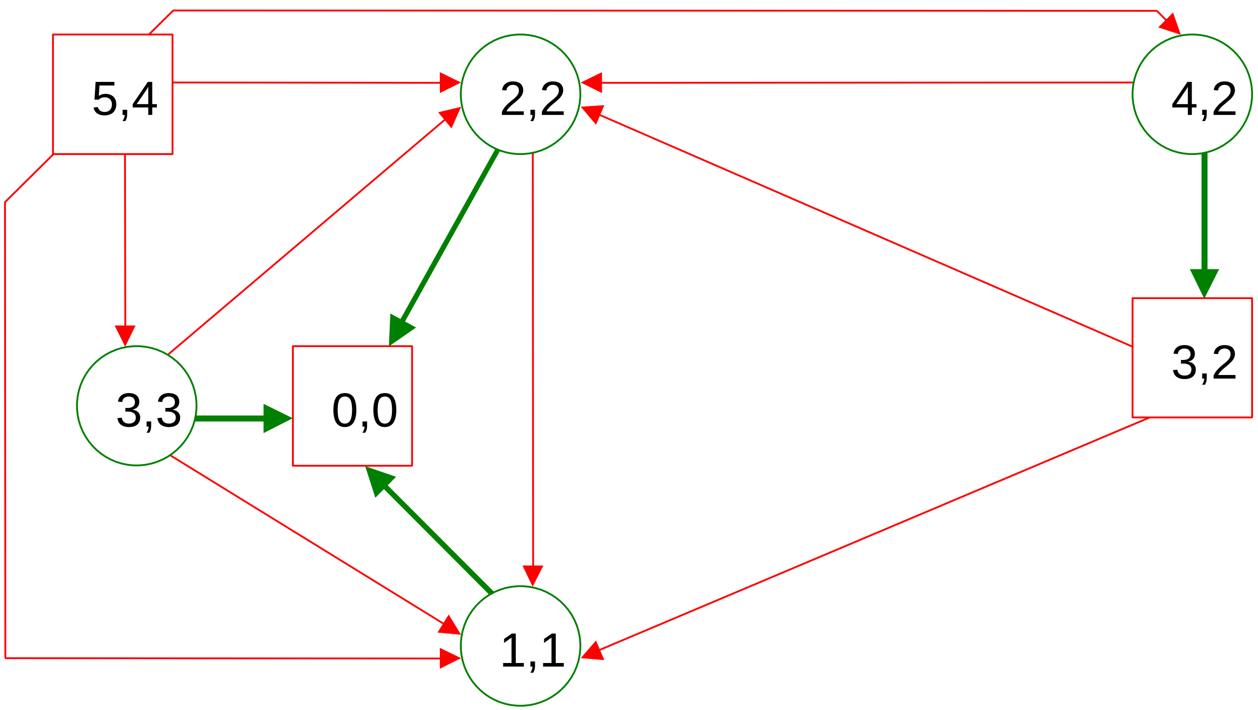
Analysing the Match Game graph for 5, 4



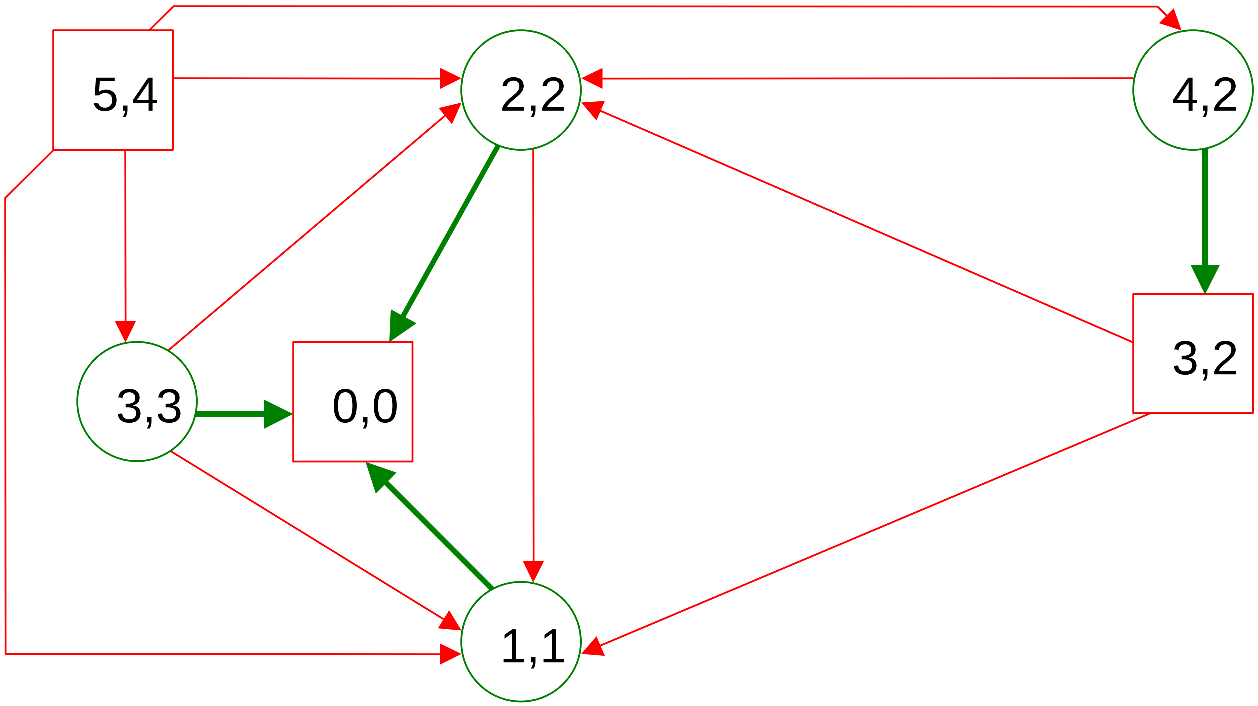
Analysing the Match Game graph for 5, 4



Analysing the Match Game graph for 5, 4



Analysing the Match Game graph for 5, 4



- So 5, 4 is a losing position.

Analysing the Match Game graph

- On a larger graph this process can be continued
- The rules can be summed up as follows:
  - A position is a winning position if at least one of its successors is a losing position.
  - A position is a losing position if all of its successors are winning positions.
- We can construct an algorithm that determines if we are at a winning position

Analysing the Match Game position

```
function recwin(i, j)
    // returns true if node i,j is winning
    for k = 1 to j do
        if not recwin(i–k, min(2k, i–k)) then
            return true
    return false
```

- This algorithm suffers the same problem as fibrec:
  - Too may recursive calls
- Can we fix this in some way?

Analysing the Match Game position

- Can we fix this in some way?
- What if we remember the nodes we have already evaluated?
- Let us construct two arrays:
  - G[0..n,0..n] where entry G[i,j] contains the value of the position i,j.
  - known[0..n,0..n] where entry known[i,j] is true if G[i,j] has been evaluated.

Analysing the Match Game position

```
  G[0, 0] = false
  known[0,0] = true
  for i = 1 to n do
     for j = 1 to i do
        known[i, j] = false

function win(i, j)
     if known[i, j] then
         return G[i, j]
     known[i, j] = true
     for k = 1 to j do
         if not win(i-k, min(2k, i–k)) then
             G(i, j) = true
             return true
     G[i, j] = false
     return false
```

Analysing the Match Game position

- This approach involves the initialisation of the array known[0..n,0..n]
- We can use virtual initialisation, described earlier, to eliminate this cost.
- This approach can be used to analyse positions in a number of games
- The basic process is the same

Analysing a general game

- – Label the terminal position(s) win, lose, draw.
- – A nonterminal position is a win if at least one of its successors is a losing position.
- – A nonterminal position is a loss if all of its successors are winning positions.
- – Any other nonterminal position is a draw.
- – Once the graph is labelled, any position can be evaluated.

Traversing a graph

– Let $G = \langle N, A \rangle$ be an undirected graph.
– We wish to visit all the nodes of G.
– We wish to do this efficiently.
– Suppose we can mark a node to show it has been visited.
– Then we can construct a recursive algorithm to visit all of the nodes as follows.
– We will use a depth first algorithm to do this.

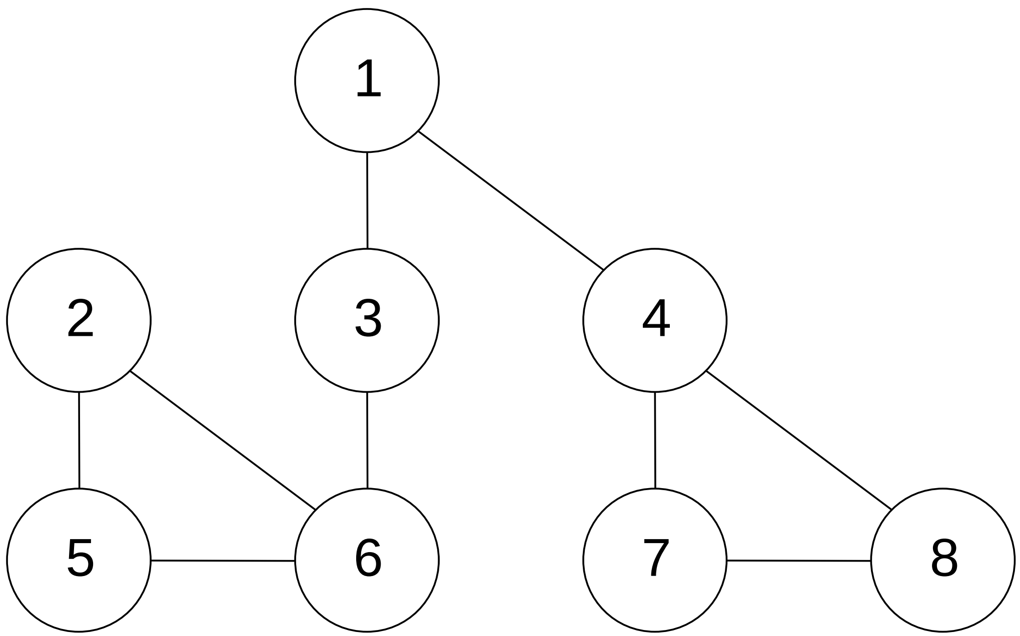
Traversing a graph

```
function dfvisit(G)
    for each v∈N do
        visited[v] = false
    for each v∈N do
        if not visited[v] then
           dfv[v]

function dfv[v]
    visited[v] = true
    for each node w adjacent to v do
       if not visited[w] then
           dfv[w]
```

Traversing a graph

– Consider the graph:



Traversing a graph

– Traversing starting at node 1

- dfv(1) initial call
-    dvf(3) recursive call
-       dfv(6) recursive call
-          dfv(2) recursive call
-             dfv(5) recursive call; progress blocked
- dfv(4) an unvisited neighbour of node 1
-    dfv(7) recursive call
-       dfv(8) recursive call; progress blocked
- There are no more nodes to visit

Traversing a graph

– Let G =⟨N, A⟩ have nnodes and aarcs (edges)

– How efficient is this algorithm?

- Since each node is visited exactly once there are ncalls to dfv.
- At each node we examine connected nodes
- This examination involves every edge being considered.

– The algorithm takes time in Θ (n) for procedure calls and a time in Θ (a) to inspect connected nodes.

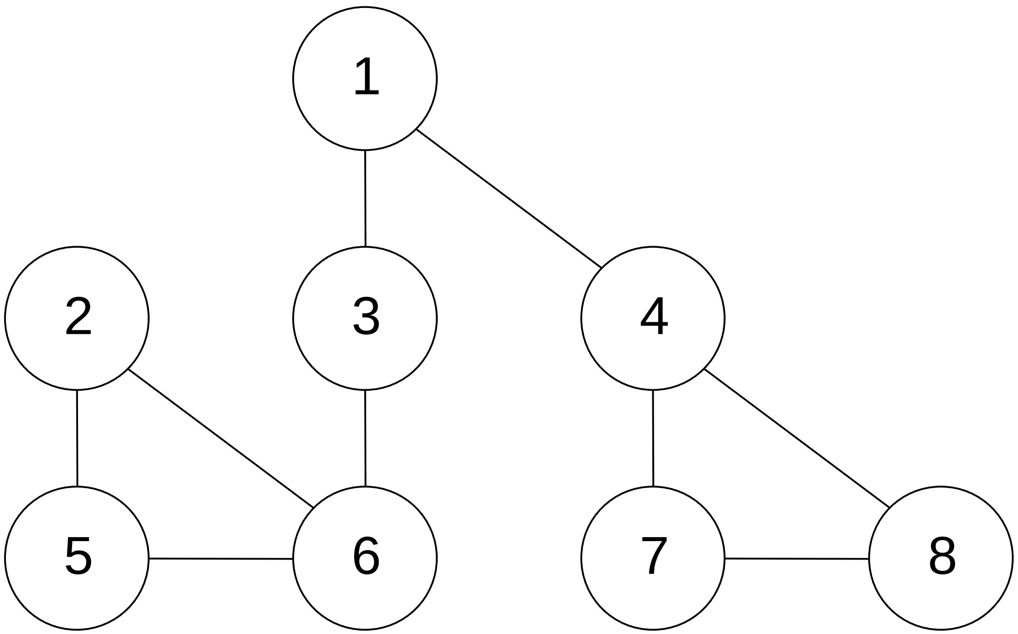
– Overall the algorithm is in Θ (max(n, a))

Traversing a graph

- Depth first traversal associates a spanning tree T with the graph.
- The edges of T correspond to the edges used to traverse the graph.
- They are directed from the first node visited to the second.
- The starting point of the traversal becomes the root of the tree.
- If the graph is not connected, we produce a forest of scanning trees.

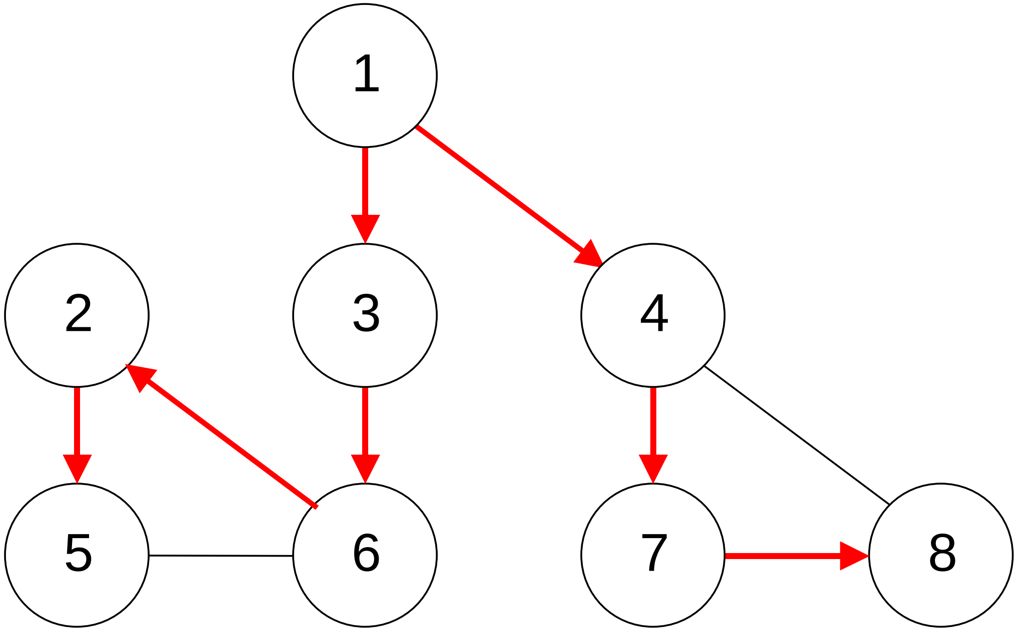
Traversing a graph

– Using our example graph:



Traversing a graph

– Using our example graph, the spanning tree is:

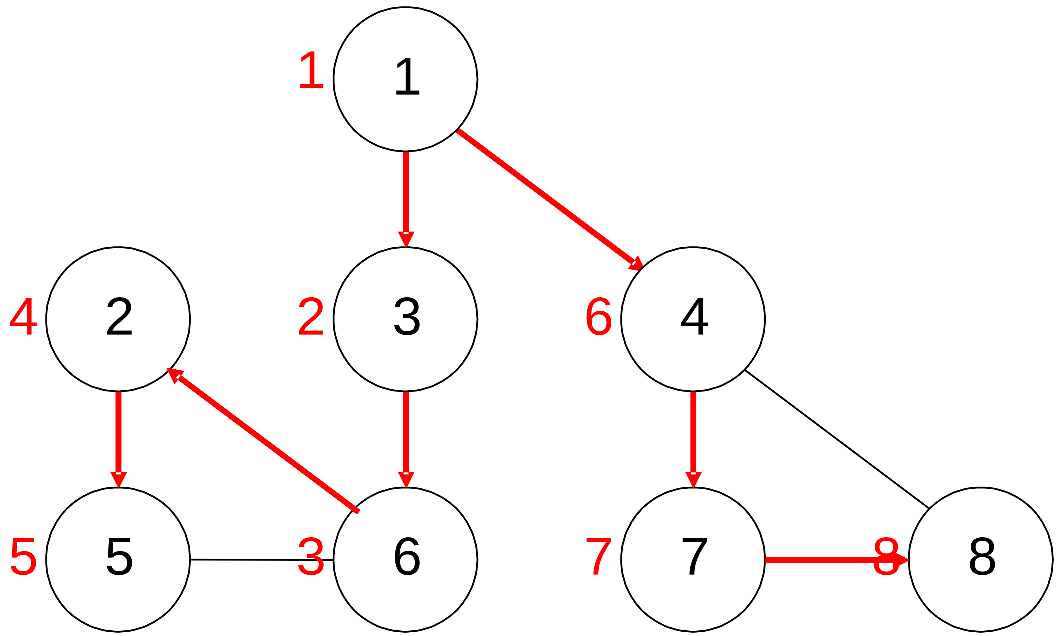


Articulation Points

- A node v of a connected graph is an articulation point if the sub graph obtained by deleting node v and all its connected edges is no longer connected.
- Node 1 and node 3 are both articulation points in the example we have been using.
- Let us label the nodes of this graph to show the preorder traversal order for T.

Articulation Points

– The preorder numbering is:

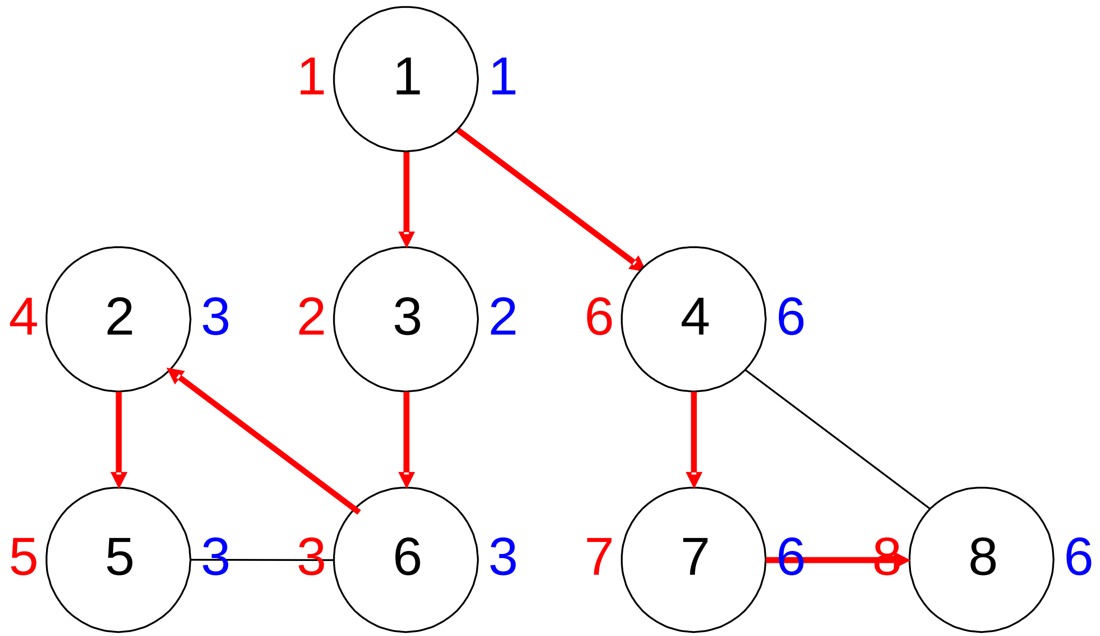


Articulation Points

- – Let us now number the graph again.
- – In this instance the number has the following property:
  - Let w be the highest node that can be reached by following down zero or more arrows and then going along at most one non-arrow edge.
- – Define prenum[w] is the serial number of w in the traversal process
- – Define highest[v] = prenum[w]
- – E.g. from node 7 we can reach node 4
  - Highest[7] = prenum[4] = 6

Articulation Points

– The highest numbering is:

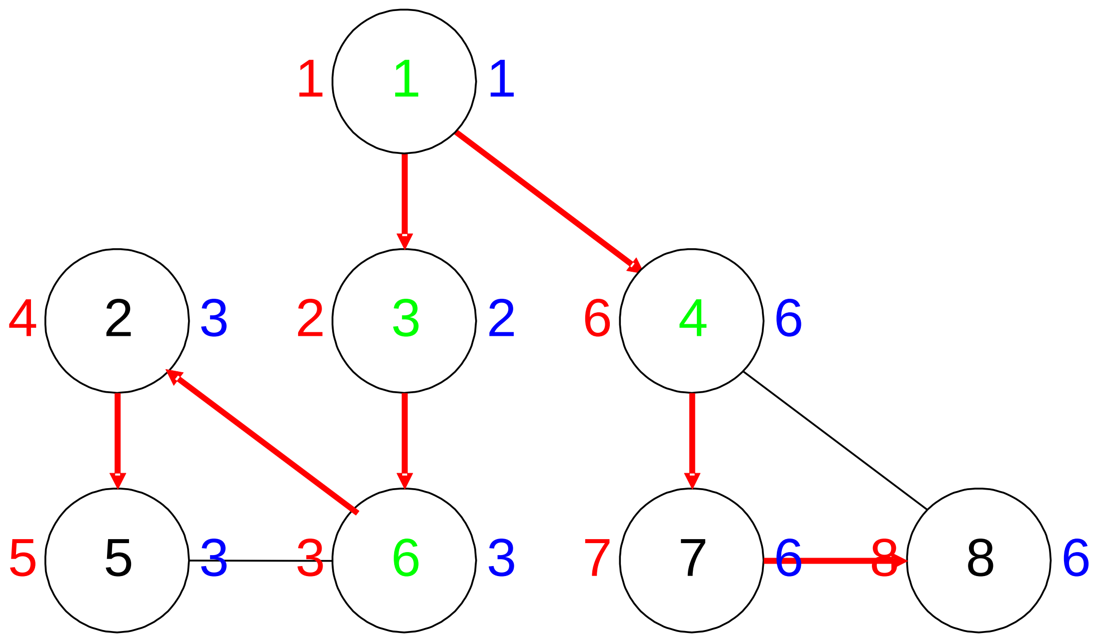


Articulation Points

- Node w must be an ancestor of v
- Consider a node v which is not the root of T
- If v has no children it cannot be an articulation point
- Otherwise let x be a child of v
  - If highest[x] < prenum[v] then we can reach a node higher up the tree than node v from node x which means that node v need not be used to reach node x.
  - If highest[x]≥prenum[v] then node x can only be reached via node v
- Node v is an articulation point if at least one of its children x satisfies highest[x] ≥ prenum[v]
- The root is an articulation point if it has more than one child.

Articulation Points

– Nodes 1, 3, 4 and 6 are articulation points.



Articulation Points

– The complete algorithm for finding the articulation points of a graph G is:

1. Conduct a depth first traversal of G, producing the spanning tree T
2. Traverse T in preorder and record the preorder sequence number prenum[v]
3. Traverse T in postorder. For each node v calculate highest[v] as the minimum of
    1. prenum[v]
    2. prenum[w] for each node w such that G contains {v, w} and T does not.
    3. highest[x] for every child x of v
4. Determine the articulation points of G as follows
    1. The root is an articulation point if it has more then one child
    2. Any other node v is an articulation point if it has at least one child x such that highest[x] > prenum[v]

Articulation Points

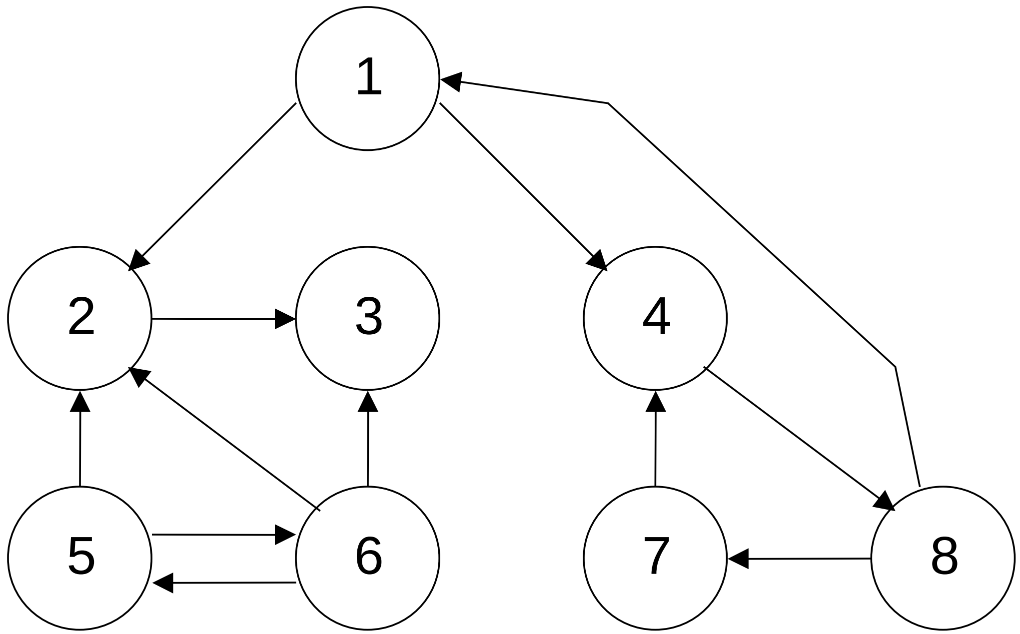
- The identification of articulation points is important in determine the critical components of networks.
- A component which corresponds to an articulation point is critical.
- If such a component fails, the network is compromised.

Depth-first Traversal: Directed Graphs

– The same algorithm can be used as for a non-directed graph.
– We only need to change the definition of adjacency to take account of the direction of each edge.
– The result of using this algorithm on a directed graph is typically quite different from its use on a non-directed graph.
– The result of the algorithm is typically a forest of sub-trees which together span the graph

Traversing a directed graph

– Consider the graph:



Traversing a directed graph

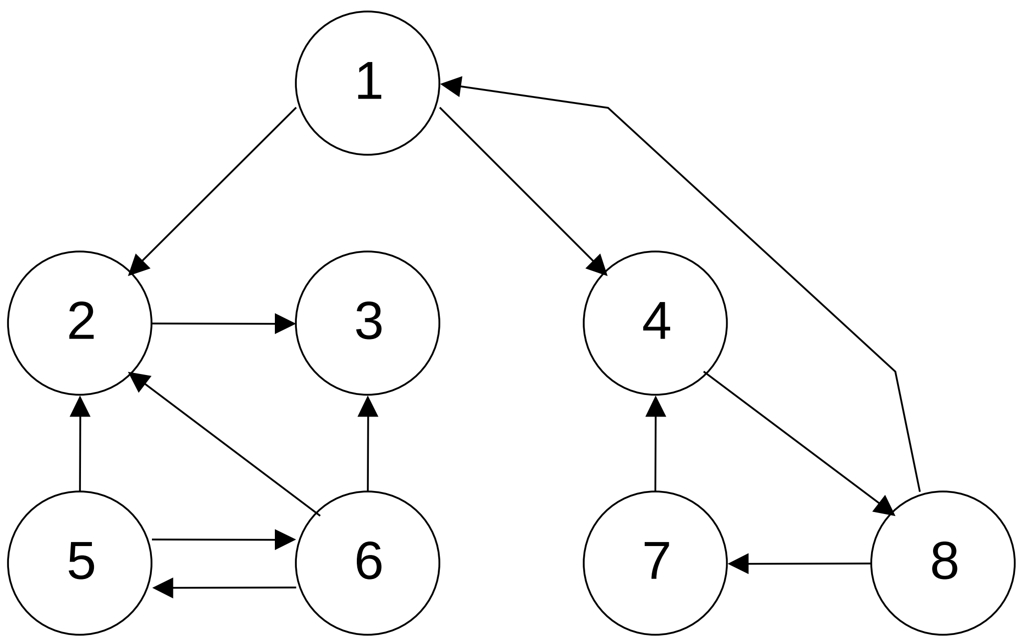
– Traversing starting at node 1

•dfv(1)	initial call
• dfv(2)	recursive call
• dfv(3)	recursive call; progress blocked
• dfv(4)	an unvisited neighbour of node 1
• dfv(8)	recursive call
• dfv(7)	recursive call; progress blocked
• dfv(5)	new starting point
• dfv(6)	recursive call; progress blocked

•There are no more nodes to visit

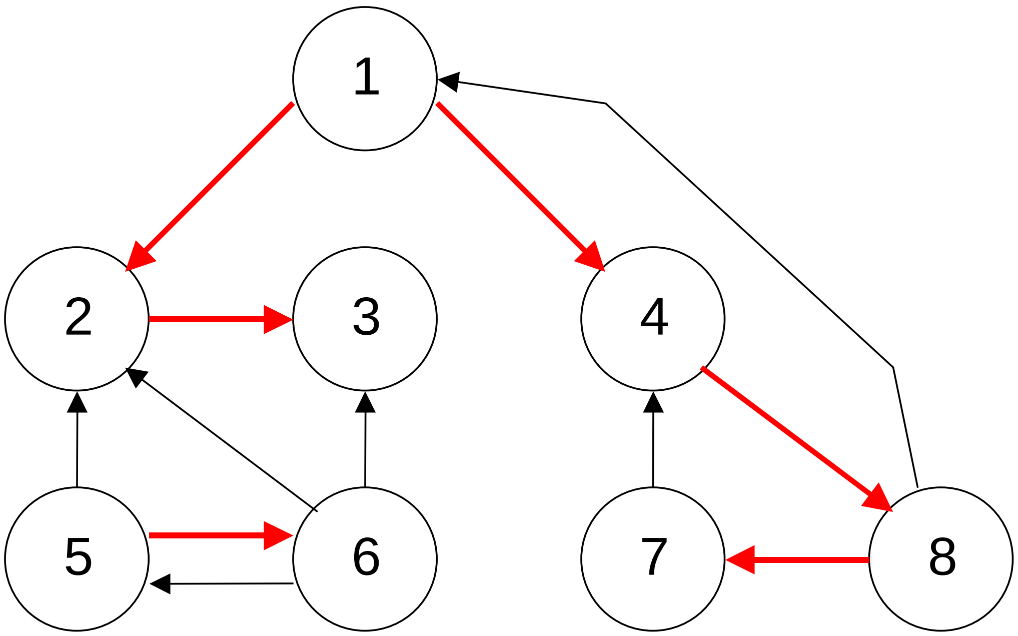
Traversing a directed graph

– Using the example graph:



Traversing a directed graph

– Using the example graph, the spanning forest *F* is:



Traversing a directed graph

– For a non-directed graph the set of edges of G not in T all connect nodes to their ancestors.

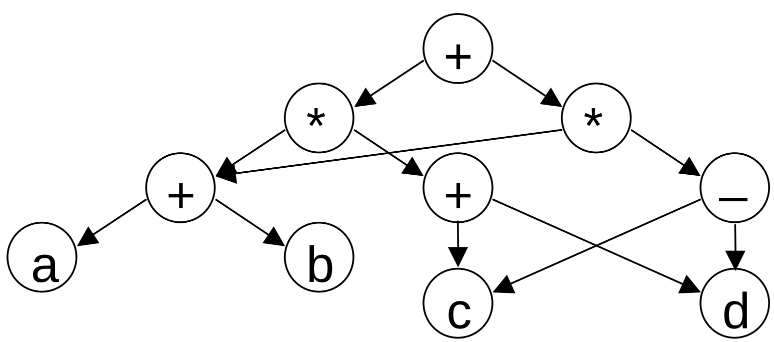
–This is not the case for edges in the directed graph G which are not in F.

–In this case, the edges fall into 3 groups:

- Those like (3, 1) which connect a node to its ancestor.
- Those like (1, 8) which connect a node to its descendant.
- Those like (5, 2) which connect two unrelated nodes.

Acyclic graphs

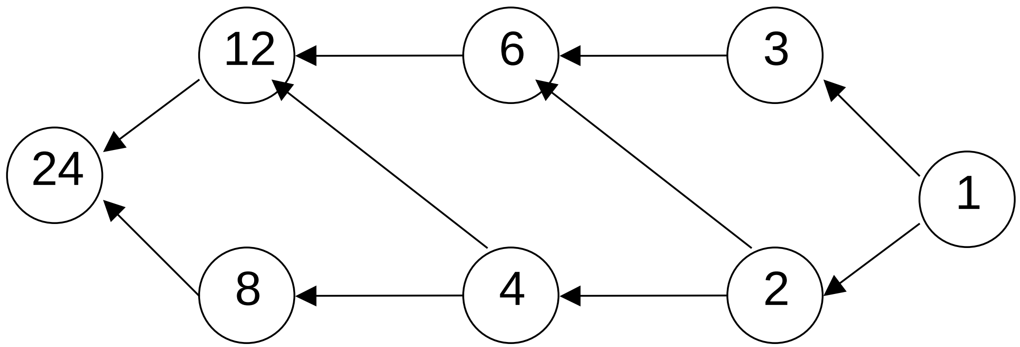
– Directed acyclic graphs can be used to represent several interesting data structures.
– For example representation of a arithmetic expression with a common sub expression.



(a + b)(c + d) + (a + b)(c – d)

Acyclic graphs

– Acyclic graphs offer a convenient way to represent partial orderings.

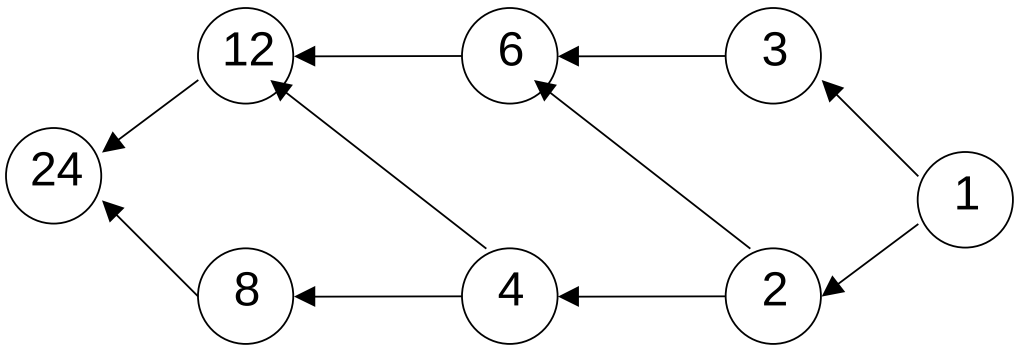


– … is a factor of.

Acyclic graphs

– Another example of a partial ordering occurs in project management where the nodes represent milestones in the project and the edges represent activities.
– A topological ordering is one in which if (i, j) is an edge then *i* comes before *j* in the ordering of the nodes.
– Using the … is a factor of graph as an example.

Acyclic graphs – Topological sorting



– Valid topological sorts are:
  - 1, 2, 3, 4, 6, 8, 12, 24
  - 1, 3, 2, 4, 6, 8, 12, 24
  - 1, 2, 3, 6, 4, 8, 12, 24
  - etc

Acyclic graphs – Topological sorting

– The depth first visit algorithm can be easily modified to do a topological sort:

```
function topsort(G)
  for each v ∈N do
     visited[v] = false
  for each v ∈N do
     if not visited[v] then
        tops[v]

function tops[v]
  visited[v] = true
  for each node w adjacent to v do
      if not visited[w] then
         tops[w]
         write v
```